Certifying Network Calculus in a Proof Assistant

Etienne Mabille, Marc Boyer†, Loïc Fejoz*, Stephan Merz‡*

**RealTime at Work, Nancy, France*
†ONERA – The French Aerospace Lab, 31055 Toulouse, France
‡Inria & LORIA, Nancy, France

Abstract

Ensuring correct behaviour of a distributed real-time function requires bounds on the network traversal time. The network calculus is a theory designed to compute such bounds, used to certify the A380 AFDX backbone. In the PEGASE project [4], some enhancements have been done to the theory, and a software implementation has been developed. To ensure correctness of the software, instead of a process based on tests and code coverage, the "proof by instance" approach is used. The tool generates a proof of correctness of the result, that can be checked by a proof assistant.

1. Result Certification for Network Calculus

In critical embedded systems such as those used in avionics, ensuring the quality of the software (using tests, qualification, and certification) represents a significant fraction of the overall cost of development. Several complementary methods exist to ensure the quality of systems, including rigorous development processes, testing of components and systems, and the use of formal methods. The latter have long been considered unable to scale up to industrial applications, and the cost of full-fledged proof remains prohibitively high in most cases. Nevertheless, advances in formal methods research have resulted in techniques that have proved practical and effective for specific problems. The Network Calculus [10] and supporting tool sets have found widespread use for determining bounds on message delays and for dimensioning buffers in embedded networks, such as in the design and certification of the Airbus A380 AFDX backbone [1, 6, 8]. However, results produced by existing tools based on the Network Calculus have to be trusted: although the underlying theory is generally well understood, implementation errors may result in faulty network designs, with unpredictable consequences.

For application domains subject to strict regulatory requirements, it appears mandatory to ensure the correctness of the results computed by the tools supporting network designs. In this contribution, we suggest a technique for certifying these results based on a "proof by instance" approach. In a nutshell (cf. Figure 1), the tool outputs a trace of the calculations it performs, as well as their results. The validity of the trace (both w.r.t. the applicability of the computation steps and the numerical correctness of the result) can be established offline by a trusted checker. For tools used at design time, we argue that this approach has several advantages over proving the calculator correct:

- Instrumenting the calculator for generating a trace is much easier (and hence less expensive) than attempting a full-fledged correctness proof, and checking the correctness of a computation is essentially trivial.
- The calculator is treated as a black box: it can be implemented using any software development process, programming language, and hardware by a tool provider separate from the checker. It can be updated without having to be requalified, as long as it still produces certifiable computation traces.
- Proof by instance is a good match for industrial processes based on testing. Nevertheless, a design that has been checked is guaranteed to be correct for any inputs matching the hypotheses of the model.

Checking the trace of a Network Calculus tool is similar to checking a mathematical proof: applying a given rule requires establishing its hypotheses, beyond pure calculation. We suggest to implement the checker by taking advantage of the trusted kernel of a proof assistant, specifically Isabelle/HOL [12]. A companion paper describing this work from the perspective of interactive theorem proving appears in [11].

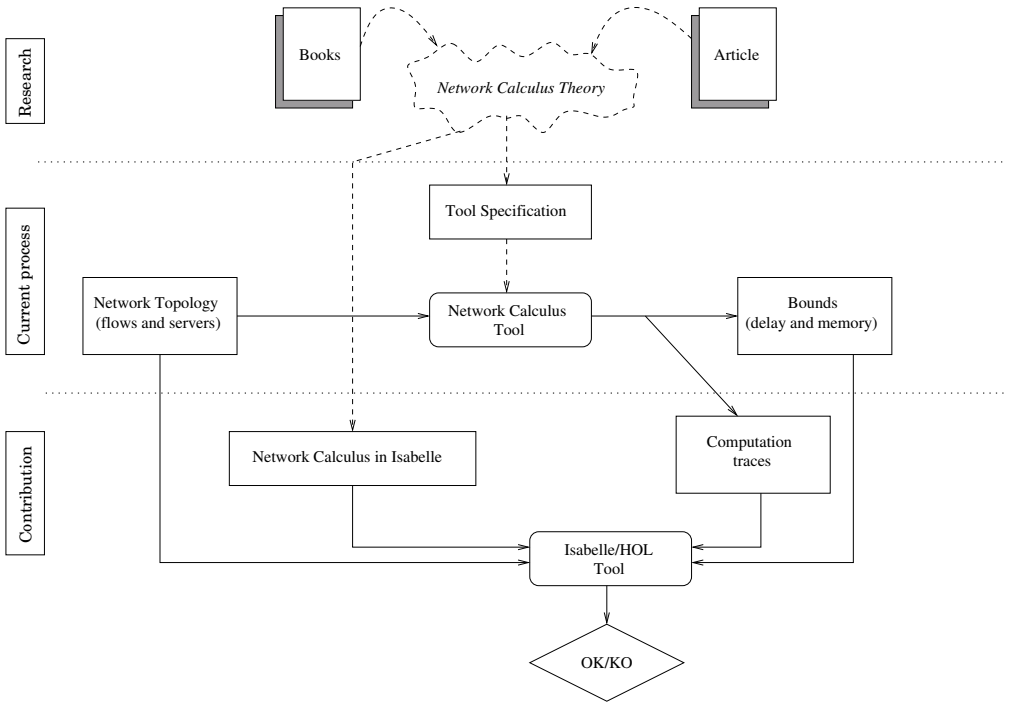

Figure 1: Proof by instance process

2. Network calculus

2.1 Mathematical background

Network calculus [10] is a theory for computing upper bounds in networks. Its mathematical background is a theory of the set of functions

$$\mathcal{F} = \{\, f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\} \mid x \leq y \implies f(x) \leq f(y) \,\} \tag{1}$$

that form a dioid under the operations $\sqcap$ and $+$ defined as pointwise minimum and addition. For practical applications, four families of functions are particularly interesting, which are defined as $\delta_d(t) = 0$ if $t \leq d$, ∞ otherwise, $\beta_{R,T}(t) = 0$ if $t \leq T$ and $R(t-T)$ otherwise, and $\gamma_{r,b}(t) = 0$ if $t \leq 0$ and $rt + b$ otherwise.

Operations of interest on $\mathcal{F}$ include convolution $\oplus$, deconvolution $\oslash$, and the sub-additive closure $f^{\oplus}$.

$$(f \oplus g)(t) = \inf_{0 \leq u \leq t} (f(t-u) + g(u)) \tag{2}$$

$$(f \oslash g)(t) = \sup_{0 \leq u} (f(t+u) - g(u)) \tag{3}$$

$$f^{\oplus} = \delta_0 \sqcap f \sqcap (f \oplus f) \sqcap (f \oplus f \oplus f) \sqcap \cdots \tag{4}$$

An important operator related to performance evaluation is the horizontal deviation $h(\cdot, \cdot)$, defined by:

$$h(f, g) = \sup_{s \geq 0} \left(\inf \{\, \tau \geq 0 \mid f(s) \leq g(s+\tau) \,\}\right) \tag{5}$$

The horizontal deviation and some common curves are illustrated on Figure 2.

These operators have important mathematical properties: for example, convolution is associative and commutative. It also "preserves" inequalities: $f \leq g \implies f \oplus h \leq g \oplus h$.

2.2 Flow and server modelling

A *flow* is represented by its cumulative function $R \in \mathcal{F}$, where $R(t)$ is the total number of bits sent by this flow up to time t. A flow R has function $\alpha \in \mathcal{F}$ as *arrival curve* (denoted $R \preceq \alpha$) if $\forall t, s \geq 0$:

Figure 2: Common curves and delay.

$R(t+s) - R(t) \leq \alpha(s)$, meaning that, from any instant t, the flow R will produce at most $\alpha(s)$ new bits of data in s time units. An equivalent condition, expressed in the $(\sqcap, +)$ dioid, is $R \leq R \oplus \alpha$.

A *server* S is a relation between an input cumulative function R, and an output cumulative function R' (denoted $R \xrightarrow{S} R'$) such that $R' \leq R$ (representing the intuition that the flow crosses the server, and that the output is produced after the input). Such a server has a *service curve* β if $R' \geq R \oplus \beta$ holds.

The maximal delay of a flow R crossing a server S is defined as the maximal horizontal deviation between the input and any possible output.

$$d(R, S) = \max_{R \xrightarrow{S} R'} h(R, R') \tag{6}$$

2.3 Some network calculus results

The delay incurred by the flow R with arrival curve α at a server S with service curve β can be bounded by the horizontal deviation between curves α and β (cf. eq. 5, and also Fig. 2).

$$d(R, S) \leq h(\alpha, \beta) \tag{7}$$

In this case, network calculus provides several results:

- $\alpha \oslash \beta$ is an arrival curve for the output flow R',
- $\delta_{d(\alpha,\beta)}$ is another service curve for the server S, and
- $\alpha \oslash \delta_{d(\alpha,\beta)}$ also is an arrival curve for the output flow R' (by simple application of the two previous items)

Having several results allows to choose one or another, depending on the context. For example, it can be shown that $\alpha \oslash \beta \leq \alpha \oslash \delta_{d(\alpha,\beta)}$, meaning that one result will produce smaller (i.e., better) numerical bounds. Conversely, the computation of $\alpha \oslash \delta_x$ is usually much simpler than that of $\alpha \oslash \beta$.

The two following results express that if α is an arrival curve for some flow R, so is $\alpha^{\oplus}$ and also any $\alpha' \geq \alpha$.

$$R \preceq \alpha \implies R \preceq \alpha^{\oplus} \tag{8}$$

$$R \preceq \alpha, \alpha \leq \alpha' \implies R \preceq \alpha' \tag{9}$$

If a flow crosses a sequence of two servers, S and S' (*i.e.* $R \xrightarrow{S} R' \xrightarrow{S''} R''$), of respective service curves β and β', then, the system made of the server sequence offers a service curve $\beta \oplus \beta'$. This result is known as the *pay burst only once phenomenon* (PBOO).

The benefit of using a formal framework is that many results have a quite simple proof, based on operator properties. For example, equation (8) can be proved using associativity of convolution: $R \leq R \oplus \alpha \leq (R \oplus \alpha) \oplus \alpha = R \oplus (\alpha \oplus \alpha) \leq \ldots \leq R \oplus (\alpha \oplus \cdots \oplus \alpha)$.

The pay burst only once result uses the same kind of proof: $R'' \geq R' \oplus \beta' \geq (R \oplus \beta) \oplus \beta' = R \oplus (\beta \oplus \beta')$.

2.4 Using network calculus to compute bounds

This presentation gives a flavor of the essence of Network Calculus: a tool box of individual theorems that can be assembled to compute results. Consider a configuration with a flow R crossing two servers S_1, S_2 in sequence: $R \xrightarrow{S_1} R' \xrightarrow{S_2} R''$. Assume that R has arrival curve α and that each server S_i offers a service of curve β_i. Then, the delay of R in S_1 can be bounded by $d = h(\alpha, \beta_1)$. Among others, $\alpha' = \alpha \oslash \delta_d$ can be chosen as an arrival curve for R'. Its sub-additive closure $(\alpha')^{\oplus}$ is also an arrival curve for R' (cf. eq. 8), but may be too expensive to compute. A simpler approximation is given by $\alpha' \sqcap \delta_0 \geq (\alpha')^{\oplus}$ (using eq. 9), and the delay of R' in S_2 can be bounded by $h(\alpha' \sqcap \delta_0, \beta_2)$. The end-to-end delay can also be bounded by the sum of bounds on local delays, *i.e.* $h(\alpha, \beta_1) + h(\alpha' \sqcap \delta_0, \beta_2)$.

Other results can be used to compute bounds: for example, the PBOO computes $h(\alpha, \beta_1 \oplus \beta_2)$ as another bound on the global delay. The implementor of an analysis algorithm will choose among the available bounds based on a compromise between the accuracy of the result, the complexity of implementation, and the required computing time.

3. The Proof Assistant Isabelle/HOL

Our objective in this work is to represent the parts of Network Calculus that underly the computation of bounds in a trusted proof assistant. Isabelle [13] is a software framework in which the syntax and proof rules of logics can be formally defined, and that provides generic tools for machine-supported reasoning. In the tradition of the LCF family of proof assistants [7], it is based on a small kernel written in the ML programming language that in particular defines a type *thm* representing theorems. Starting from user-defined axioms and proof rules specific to the particular logic, new theorems can only be created by applying rules to already existing theorems. In this way, the soundness of derived theorems only depends on that of the user-provided axioms and proof rules (which can easily be inspected) and the correct implementation of the kernel that implements rule application. The Isabelle framework provides many semi-automatic tools for proof search, such as a tableau-based theorem prover, a rewriting engine or decision procedures for fragments of arithmetic, and these tools can be instantiated for different logics. Moreover, users can program new proof methods (or tactics) from scratch or reusing the generic tools. All these methods provide justifications that are checked by the Isabelle kernel, and only then the result is accepted as a theorem. In this sense, result certification is extensively used within the Isabelle framework itself.

Isabelle/HOL [12] is the encoding in Isabelle of higher-order logic, an expressive language for formalizing mathematical theories. HOL is a typed logic whose type system is close to that of standard functional programming languages. It provides base types such as *bool*, *nat* or *real*, function and product types such as $real \Rightarrow bool$ or $nat \times nat$, type constructors including lists or sets such as $(real \times bool)\ set$, and type variables written $'a$, $'b$ etc. The function type constructor associates to the right, so $real \Rightarrow real \Rightarrow bool$ is the type of functions that expect two real-valued arguments and return a boolean value. Function application is denoted by juxtaposition, and functions are first-class objects in HOL: if f is a function of the above type and r is an expression of type *real*, then $f\ r$ denotes a unary function of type $real \Rightarrow bool$. For any expression, Isabelle will infer its most general type, and type annotations are therefore usually unnecessary. They may nevertheless be used for purposes of documentation or for restricting an expression to a less general type, as in $0 :: nat$.

New types can be introduced as algebraic data types based on constructors, as in functional programming. For example, a data type of trees could be defined with constructors *Leaf* and *Node* that represent a leaf and an inner node of a tree. Isabelle/HOL also provides the **typedef** construct for introducing a type based on a characteristic predicate. For example,

typedef $even = \{n :: nat\ .\ \exists m.\ n = m + m\}$

introduces a type isomorphic to the set of even natural numbers. Since types must be non-empty in HOL for logical consistency, such definitions come with a proof obligation that requires the user to prove the existence of some value satisfying the predicate.

Developments in Isabelle are organized in *theory files* that contain definitions of types and operators as well as statements and proofs of lemmas and theorems. For example,

definition *is-upper-bound* :: $real\ set \Rightarrow real \Rightarrow bool$ **where**
$is\text{-}upper\text{-}bound\ S\ r\ \equiv\ \forall x \in S.\ x \leq r$

introduces a predicate that indicates if a real number is an upper bound of a set of real numbers. Isabelle/HOL also supports operators defined inductively, as in the following definition of binomial coefficients

fun *binom* $n\ 0 = 1$
| *binom* $0\ (Suc\ k) = 0$
| *binom* $(Suc\ n)\ (Suc\ k) = binom\ n\ k + binom\ n\ (Suc\ k)$

where *Suc* denotes the successor function over natural numbers. Since HOL is a logic of total functions, inductive definitions must be shown to be terminating. In many cases, such as for the above function *binom*, this termination proof can be found automatically by built-in heuristics.

Lemmas and theorems assert facts, typically involving previously defined operators. For example, we could state the following lemma about upper bounds:

lemma *upper-bound-union*:
assumes *is-upper-bound* $S\ r$ **and** *is-upper-bound* $T\ r$
shows *is-upper-bound* $(S \cup T)\ r$

All variables that appear in the statement of a theorem are implicitly universally quantified. The user must provide a proof that can be checked using Isabelle's proof methods. For a simple lemma such as the one above, one would simply expand the definition of the operator *is-upper-bound*, then invoke Isabelle's default automatic prover. In more complex cases, proofs could proceed by induction, possibly based on a generalization of the statement of the theorem.

Finally, Isabelle/HOL also provides facilities for generating executable code in functional programming languages (including ML, Haskell, and Scala) from operator definitions. Programs generated in this way enjoy a high degree of trustworthiness since they are based on the same definitions that underly formal proofs about these operators. Although we do not currently rely on code generation for our result certifier, we may consider doing so in the future, since a stand-alone program can execute much more efficiently than when all computations are checked by the Isabelle kernel, as is the case for our current prototype.

4. Encoding Network Calculus in Isabelle

4.1 Mathematical Bases

In order to develop a result certifier within Isabelle, we need to formalize the theory underlying Network Calculus, to the extent that it underlies the algorithms whose results we intend to certify. As a side benefit of this formalization, we obtain a rigorous development of Network Calculus, including all possible corner cases that may be overlooked in pencil-and-paper proofs. We now give an outline of our formal development.

The fundamental notion in Network Calculus is that of a *flow*, modeled as a non-decreasing function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. We define a suitable type in Isabelle as

typedef $ndf = \{f :: ereal \Rightarrow ereal\ .\ (\forall r \leq 0.\ f\ r = 0) \wedge mono\ f\}$

where *ereal* is a pre-defined type corresponding to $\mathbb{R} \cup \{\infty\}$, and *mono f* holds if f is a weakly monotonic function. Note that we extended the domain of f to arbitrary real numbers, fixing the result to be zero over negative reals, as this turned out to simplify the subsequent definitions. Over this function type, we define operations such as $+$, $*$, and $\leq$ by pointwise extension over the arguments, and establish basic algebraic properties: for example, the resulting structure forms an ordered commutative monoid with 0 and 1.

We also introduce specific classes of functions such as $\delta_d(t)$, $\beta_{R,T}(t)$ and $\gamma_{r,b}(t)$ mentioned in Section 2.1 and characteristic operations on flows. For example, the operation of *convolution* introduced in equation (2) is represented in Isabelle as

definition *convol* :: $ndf \Rightarrow ndf \Rightarrow ndf$ (**infix** $\oplus$) **where**
$f \oplus g \equiv Abs\text{-}ndf\ (\lambda t.\ if\ t < 0\ then\ 0$
$else\ Inf\ \{f \cdot (t-u)\ +\ g \cdot u \mid u.\ 0 \leq u \wedge u \leq t \wedge u \neq \infty\})$

where *Abs-ndf* denotes the injection from functions over type *ereal* to flows of type *ndf*, and where $\cdot$ denotes the evaluation of a flow at an argument of type *ereal*.

In our Isabelle theories we prove characteristic properties of flows and operations. For example, a flow is called *sub-additive* if its value at argument $x + y$ is at most the sum of its values at x and y, formally

definition *is-sub-additive* :: $ndf \Rightarrow bool$ **where**
$is\text{-}sub\text{-}additive\ f \equiv (\forall x\ y.\ 0 \leq x \wedge 0 \leq y \longrightarrow f \cdot (x+y) \leq f \cdot x\ +\ f \cdot y)$

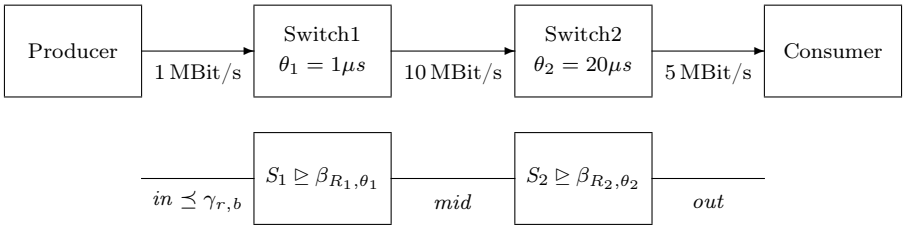

Figure 3: A simple system and its Network Calculus representation.

and we prove that the convolution of two sub-additive flows is again sub-additive:

lemma *convol-sub-add-stable*:
assumes *is-sub-additive f* **and** *is-sub-additive g*
shows *is-sub-additive* $(f \oplus g)$

Similar definitions and corresponding properties are formalized for the other fundamental constructs of Network Calculus, mirroring the informal presentation in Section 2.1.

4.2 Servers and service curves

Network Calculus represents a *simple server* as a left-total relation between (input and output) flows such that the output flow is not larger than the input flow:

typedef $server = \{s :: (ndf \times ndf)\ set.\ (\forall in.\ \exists out.\ (in, out) \in s) \wedge (\forall (in, out) \in s.\ out \leq in)\}$

and we define what it means for a flow to be constrained by an arrival curve α and for a server to provide minimum service β:

$$R \preceq \alpha \ \equiv\ R \leq R \oplus \alpha \qquad S \trianglerighteq \beta \ \equiv\ \forall (in, out) \in S : in \oplus \beta \leq out.$$

The delay between two flows f and g at argument t corresponds to the least τ such that the flow g evaluated at time $t + \tau$ exceeds flow f at time t:

definition *delay* :: *ndf* ⇒ *ndf* ⇒ *eral* ⇒ *ereal* **where**
$delay\ f\ g\ t \ \equiv\ Inf\ \{\tau.\ 0 \leq \tau \wedge f \cdot t \leq g \cdot (t + \tau)\}$

and the maximal delay of a flow R crossing server S, noted $d(R, S)$ in equation (6), is the maximal delay between the input R and any possible output of the server.

definition *worst-delay-server* :: *ndf* ⇒ *server* ⇒ *ereal* **where**
$worst\text{-}delay\text{-}server\ R\ S \ \equiv\ Sup\ \{\ delay\ R\ R'\ t \mid t\ R'.\ t \neq \infty \wedge (R, R') \in Rep\text{-}server\ S\ \}$

Again, we prove results relating these constraints to bounds on delays and backlogs. For example, the following theorem, corresponding to equation (7) of Section 2.3, provides a bound on the delay of a simple server

theorem *d-h-bound*:
assumes $in \preceq \alpha$ **and** $S \trianglerighteq \beta$
shows $worst\text{-}delay\text{-}server\ in\ S \leq h\text{-}dev\ \alpha\ \beta$

where the horizontal deviation *h-dev* is the name of the function defined in equation (5).

Moreover, we define constructs such as composing servers in sequence or packetization. Finally, these concepts are extended to multiple-input multiple-output servers that take vectors of flows as input and output.

5. Certifying a Simple Network Computation

In order to illustrate the use of our theories on a simple example, let us consider the producer-consumer setup shown in Fig. 3. The producer sends at most one frame every $T = 20\,\text{ms}$. We assume that the payload is of maximal size 980 bytes. Assuming an overhead of 20 bytes per frame, the maximum frame size is $MFS = 8000\,\text{bits}$. This flow is sent to a consumer via two switches with switching delays $\theta_1 = 1\,\mu s$ and $\theta_2 = 20\,\mu s$. The physical links between the producer, the switches, and the consumer have bandwidths of 1, 10 and 5 MBit/s, respectively.

The NC model appears in the lower part of Fig. 3. Flow *in* is constrained by the arrival curve $\alpha_{in} = \gamma_{r,b}$ where b equals MFS and $r = \frac{MFS}{T} = \frac{8000}{20\times10^3} = \frac{2}{5}$.

The service curves are given by the function β_{R_i,θ_i} where the bandwidths are $R_1 = 10\,\text{bit}/\mu s$ and $R_2 = 5\,\text{bit}/\mu s$, and the delays are θ_1 and θ_2.

We are interested in the maximal delays that frames may incur. Using theorem *d-h-bound*, the delay at server 1 is bounded by $h(\alpha_{in}, \beta_{10,1})$, which evaluates to 801 μs. As explained in Section 2, the arrival curve of flow *mid* can be computed as

$$\alpha_{mid} = (\alpha_{in} \oslash \delta_{801}) \sqcap \delta_0 = (\gamma_{\frac{2}{5},8000} \oslash \delta_{801}) \sqcap \delta_0 = \gamma_{\frac{2}{5},\frac{41602}{5}}\,.$$

Continuing for the second server, its delay is at most $h(\alpha_{mid}, \beta_{5,20}) = \frac{42102}{25}\,\mu$s. Consequently, the overall delay incurred by frames equals

$$801\,\mu\text{s} + \frac{42102}{25}\,\mu\text{s} = \frac{62127}{25}\,\mu\text{s}.$$

These computations are performed by a Java prototype and certified using Isabelle.

The trace shown in the appendix A consists of output of the prototype interleaved with Isabelle lemmas whose assertions and proofs were automatically generated.

6. Industrial effort

Formal methods are often presented as a way to increase software quality. We believe that they can also contribute to decreasing the costs of software development in case the software has to meet high standards of qualification.

The avionics industry has a long experience of ensuring high quality software: the DO-178B (and the new DO-178C) standards guide the development process, aiming at increasing quality by imposing rigorous requirements on development, documentation, verification, and configuration. Enforcing these requirements has a significant cost: following a stringent process increases the cost of software production by a factor that is estimated between 4 and 10. We expect that proof by instance can significantly decrease the cost of producing a tool set for computing bounds on network delays, implementing network calculus.

Such a tool is mainly composed of two elements:

1. a $(\min, +)$ library, implementing the basic operations on functions (including $\min, +, \oplus, \oslash, \cdot^{\oplus}$, $h(\cdot,\cdot)$, etc.) and,
2. a network calculus engine, applying network calculus theorems on subsystems and producing numerical results.

Each component can have different implementations, handling more complex and accurate functions and algorithms.

From the COINC project [5] we know that it takes about 1 man-year to develop a state of the art $(\min, +)$ library, implementing the very generic algorithms of [2]. Developing a full AFDX analyser based on network calculus represents an effort of about 3 man-years (including the $(\min, +)$ library), based on our experience on the RTaW-PEGASE tool [3].

To evaluate the feasibility of proof by instance for a network calculus tool, a prototype was developed: this prototype, presented in the previous section, implements a simple version of the $(\min, +)$ library and some elementary network calculus algorithms.

The development was carried out during a 9-month internship. This effort includes the development of the network calculus engine, but also the modelling of a network calculus library in Isabelle (cf. "Network calculus in Isabelle" box in Figure 1), and trace generation (cf. "Computation traces" box in Figure 1).

Since the $(\min, +)$ library and the network calculus algorithms used in the current version are quite basic, the accuracy of the tool is not very good: it compute bounds that are, globally, two times bigger than a state of the art tool, such as the RTaW-PEGASE tool [3]. Moreover, the Isabelle library is not fully proved: some results are presently considered as axioms, and we plan to fully prove them, further increasing our confidence in the correctness of the calculations.

Nevertheless, the low effort required for the current prototype confirms our intuition that proof by instance is a good approach for producing high-quality software at a reasonable cost. The quality of the result is ensured by the Isabelle kernel that implements conceptually simple operations and underlies many significant formal developments [9].

7. Conclusion

We argue that the computations of tool sets used in the applications of formal methods such as Network Calculus should be certified in order to increase the confidence in the correctness of the designs. The work reported here attempts to evaluate the feasibility of such certification, based on a formalization of Network Calculus in the interactive proof assistant Isabelle/HOL.

Developing a Network Calculus engine that is able to handle an AFDX configuration requires about three years of implementation. The effort for developing a qualified version of such an engine, using state-of-the-art techniques (documentations, testing, peer-review, etc.) is higher by a factor of 4 or 10.

The proof-by-instance approach promises to reduce this effort while increasing the confidence in the results produced by the software. We have so far invested less than 1 development year for encoding the fundamental concepts of Network Calculus in Isabelle, and for instrumenting an existing tool to produce a full formal proof of the correctness of bounds for one single switch. We estimate that the overall effort for producing the proof for a realistic network should be between 2 and 3 years. In particular, it will be important to speed up the computations on real numbers inside the proof assistant.

In other words, we believe that result certification could lower the overhead for developing a trustworthy version of a Network Calculus tool to a factor of 2 or 3, while significantly improving its quality.

References

[1] AEEC. Arinc 664p7-1 aircraft data network, part 7, avionics full-duplex switched ethernet network. Technical report, Airlines Electronic Engineering Committee, september 2009.

[2] Anne Bouillard and Éric Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 17(4), october 2007. http://www.springerlink.com/content/ 876x51r6647r8g68/.

[3] Marc Boyer, Jörn Migge, and Marc Fumey. PEGASE, a robust and efficient tool for worst case network traversal time. In *Proc. of the SAE 2011 AeroTech Congress & Exhibition*, Toulouse, France, 2011. SAE International.

[4] Marc Boyer, Nicolas Navet, Xavier Olive, and Eric Thierry. The PEGASE project: Precise and scalable temporal analysis for aerospace communication systems with network calculus. In Tiziana Margaria and Bernhard Steffen, editors, *4th Intl. Symp. Leveraging Applications (ISoLA 2010)*, volume 6415 of *LNCS*, pages 122–136, Heraklion, Greece, 2010. Springer. `http://www.realtimeatwork.com/software/rtaw-pegase/`.

[5] ARC COINC home page. `http://perso.bretagne.ens-cachan.fr/~bouillar/coinc/`.

[6] Fabrice Frances, Christian Fraboul, and Jérôme Grieu. Using network calculus to optimize AFDX network. In *Proceeding of the 3thd European congress on Embedded Real Time Software (ERTS'06)*, Toulouse, January 2006.

[7] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, Heidelberg, 1979.

[8] Jérôme Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, Institut National Polytechnique de Toulouse (INPT), Toulouse, Juin 2004.

[9] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Comm. ACM*, 53(6):107–115, 2010.

[10] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus*. Springer, 2001.

[11] Etienne Mabille, Marc Boyer, Loïc Fejoz, and Stephan Merz. Towards certifying network calculus. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, Rennes, France, 2013. Springer.

[12] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer Verlag, 2002.

[13] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *21st Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38, Montréal, Canada, 2008. Springer.

A. Example Trace

```
1  ##################################
2  # Used algorithm: SFA FIFO
3  ##################################
4  # Time unit: microsecond
5  # Frame size unit: bit
6  ##################################
7  delta0 := delay(0)
8  ##################################
9  # Input flows
10 ##################################
11 # unique_flow entering at System_1>port-port1
12 lmax_unique_flow := 8000
13
14 ##################################
15 # EndSystem : System_1>port-port1
16 ##################################
17 # Latency in transmission: 'internal delay'
18 id_unique_flow_System1portport1 := 0
19 # Packet arrival curve for unique_flow on System_1->theRouter
20 unique_flow_System1portport1_P := star(uaf([(0,0)]](0,8000)2/5(+Infinity,+Infinity)[])
21
22 assert(unique_flow_System1portport1_P = uaf([(0,0)]](0,8000)2/5(+Infinity,+Infinity)[])
```

lemma *AC_sub_add_closure: "sub_add_closure (gamma (2/5) 8000) = gamma (2/5) 8000"*
using *sub_add_closure_gamma* **by** *auto*

```
1  ##################################
2  # Server : theRouter>port-P3
3  ##################################
4  # Cumulated arrivals
5  cumA_theRouterportP3 := zero
6  # unique_flow on System_1->theRouter
7  cumA_theRouterportP3 := cumA_theRouterportP3 + unique_flow_System1portport1_P
8  assert(cumA_theRouterportP3 = uaf([(0,0)]](0,8000)2/5(+Infinity,+Infinity)[])%
```

lemma *AC_add_zero: "(gamma (2/5) 8000) + (0::ndf) = gamma (2/5) 8000"* **by** *auto*

```
1  # Common service
2  S_theRouterportP3 := uaf([(0,0)0(1,0)]](1,0)10(+Infinity,+Infinity)[)
3  # Common delay
4  d_theRouterportP3:=hDev(cumA_theRouterportP3, S_theRouterportP3)
5
6  assert(d_theRouterportP3 = 801)%
```

lemma *delay_server1:*
assumes *"(R, R') ∈ Rep_server S"* **and** *"S ⊵ beta 10 1"* **and** *"R ≺ gamma (2/5) 8000"*
shows *"worst_delay_server R S ≤ ereal(801)"* **proof** -
  from *assms d_h_bound* **have** *"worst_delay_server R S ≤ h_dev (gamma (2/5) 8000) (beta 10 1)"*
    by *auto*
  with *hor_dev_beta_gamma* **show** *"worst_delay_server R S ≤ ereal (801)"* **by** *auto*
qed

```
# unique_flow on theRouter->theSecondRouter
# residual service
rS_unique_flow_theRouterportP3 := delay(d_theRouterportP3)
# induced arrival curve
unique_flow_System1portport1_theRouterportP3 := unique_flow_System1portport1_P /
    rS_unique_flow_theRouterportP3
unique_flow_System1portport1_theRouterportP3 := unique_flow_System1portport1_theRouterportP3 /\
    delta0

assert(unique_flow_System1portport1_theRouterportP3 =
    uaf([(0,0)]](0,41602/5)2/5(+Infinity,+Infinity)[))%
```

lemma *out_ac_server1:*
assumes *"(R, R') ∈ Rep_server S"* **and** *"S ⊵ beta 10 1"* **and** *AC_in: "R ≺ gamma (2/5) 8000"*
shows *"R' ≺ gamma (2/5) (41602/5)"* **proof** -
  have *R'_cont: "continuous_on (range ereal) (Rep_ndf R')"* **sorry**
  have *"worst_delay_server R S ≤ ereal 801"* **using** *assms delay_server1* **by** *auto*
  with *Out_AC_Delay* **have** *"R' ≺ gamma (2 / 5) 8000 ⌾ delta (ereal 801)"*
    using *assms gamma_real PInfty_neq_ereal(1) R'_cont* **by** *auto*
  thus *?thesis* **using** *deconv_gamma_delta[of "ereal 801" "2/5" "8000"]* **by** *auto*
qed

```
# packetization
unique_flow_System1portport1_theRouterportP3_P := unique_flow_System1portport1_theRouterportP3


##################################
# Server : theSecondRouter>port-P2
##################################
# Cumulated arrivals
cumA_theSecondRouterportP2 := zero
# unique_flow on theRouter->theSecondRouter
cumA_theSecondRouterportP2 := cumA_theSecondRouterportP2 +
    unique_flow_System1portport1_theRouterportP3_P

assert(cumA_theSecondRouterportP2 = uaf([(0,0)]](0,41602/5)2/5(+Infinity,+Infinity)[))%
```

lemma *AC'_add_zero: "gamma (2/5) (41602/5) + (0::ndf) = gamma (2/5) (41602/5)"* **by** *auto*

```
# Common service
S_theSecondRouterportP2 := uaf([(0,0)0(20,0)]](20,0)5(+Infinity,+Infinity)[)
# Common delay
d_theSecondRouterportP2:=hDev(cumA_theSecondRouterportP2, S_theSecondRouterportP2)

assert(d_theSecondRouterportP2 = 42102/25)%
```

lemma *delay_server2:*
assumes *"(R', R'') ∈ Rep_server S2"* **and** *"S2 ⊵ beta 5 20"* **and** *"R' ≺ gamma (2/5) (41602/5)"*
shows *"worst_delay_server R' S2 ≤ ereal(42102/25)"* **proof** -
  from *assms d_h_bound*
  have *"worst_delay_server R' S2 ≤ h_dev (gamma (2/5) (41602/5)) (beta 5 20)"* **by** *auto*
  with *hor_dev_beta_gamma* **show** *"worst_delay_server R' S2 ≤ ereal (42102/25)"* **by** *auto*
qed

```
##################################
# WCTT
##################################
# Bound for unique_flow received by System_2
# internal delay in sending server System_1>port-port1
```

```
b_unique_flow_System_2 := 0
# delay induced by theRouter>port-P3
b_unique_flow_System_2:= b_unique_flow_System_2 + d_theRouterportP3
# delay induced by theSecondRouter>port-P2
b_unique_flow_System_2:= b_unique_flow_System_2 + d_theSecondRouterportP2

assert(b_unique_flow_System_2 = 62127/25)%
```

lemma *"0 + ereal (801)+ ereal (42102/25) = ereal (62127/25)"* **by** *auto*

end